Freeform Search

Database:
US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Term: 6826580.pn.

Display: 50 **Documents in Display Format:** FRO **Starting with Number** 1

Generate: ○ Hit List ◉ Hit Count ○ Side by Side ○ Image

Search Clear Interrupt

Search History

DATE: Friday, November 18, 2005 Printable Copy Create Case

10/103,415

6,606,301

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=OR			
L28	6826580.pn.	1	L28
L27	L26 and ("file system")	5	L27
L26	L25 and (client same server)	28	L26
L25	(disk and (formal near description))	109	L25
L24	L23 and (formal near description)	0	L24
L23	client near "file system"	273	L23
L22	112 and server	0	L22
L21	"self-describing" same (file near system)	4	L21
L20	L19 and block$	28	L20
L19	L18 and SAN	37	L19
L18	(file near system) and ("self-describing")	171	L18
L17	L15 and (shared near access)	0	L17
L16	L15 and SAN	0	L16
L15	L14 and ("self-describing")	1	L15
L14	5640559.pn.	1	L14
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			

L13	L10 and structure	15	L13
L12	L10 and inode	0	L12
L11	L10 and (block near allocation)	0	L11
L10	L9 and block	15	L10
L9	L8 and server	15	L9
L8	L7 and client	15	L8
L7	L6 and disk	18	L7
L6	L5 and (formal near description)	25	L6
L5	SAN and (file near system)	6651	L5
L4	L3 and (formal near description)	2	L4
L3	L2 and (disk near access$)	659	L3
L2	L1 and SAN	6651	L2
L1	file near system	39440	L1

END OF SEARCH HISTORY

L20: Entry 24 of 28 File: USPT Sep 1, 1998

DOCUMENT-IDENTIFIER: US 5802365 A
TITLE: Dynamic device matching using driver candidate lists

Inventor City (2):
San Jose

Inventor City (4):
San Jose

Inventor Group (2):
Saulpaugh; Thomas E. San Jose CA

Inventor Group (4):
Banks; Jano San Jose CA

Brief Summary Text (6):
In the past, devices were matched with their proper driver by strict one to one correspondence that was typically manually performed by the computer system user. That is to say, the computer system user would alter the contents of a system file that was read at computer "boot" and this system file would contain a list of drivers that the computer system recognized and would associate a particular driver to a particular device according to an inflexible listing. The driver first needed to be loaded into the computer system before the system file was updated by the computer user so that the system would recognize the driver. This system is referred to herein as "hard coding" the drivers to their associated devices. While workable in some respects for knowledgeable computer users, this system is undesirable for computer users that do not have the know how to perform the proper matching between a given device and its driver or for those that do not know the location of the proper driver. It would be desirable, then, to provide a mechanism and method for reducing problems associated with configuring the proper driver with its associated device in a computer system. The present invention provides such advantageous solution.

Drawing Description Text (2):
FIG. 1 is a block diagram illustration of a computer system used by embodiments of the present invention.

Detailed Description Text (15):
Under the present invention, native device drivers 80 (or just "drivers") are stored in several locations within the computer system 120. A driver 80 (FIG. 5a) can be located in RAM 102, ROM 103 (e.g., within expansion ROM located on the device itself), in a file system (e.g., on a disk drive 104), or may be directly located within a device node 10a-10k in the device tree 10 database. In the latter case device matching is not typically required for a device node having the driver associated therewith unless a more compatible driver is located elsewhere. For the majority of cases the device node does not have the driver associated with it in the device tree 10. The present invention automatically matches up a device of the device tree 10 with its appropriate driver. The drivers located in the device tree 10 are sometimes called default drivers and are said to exist within device driver property for the node. Drivers located in the file system 104 are referred to as

drivers in a "device driver folder" and can override the default driver under the present invention by use of candidate list matching and candidate list priority sorting as described to follow.

Detailed Description Text (25):
To determine the kind of request or kind of command, the Device Manager 95 parameter block has procedure parameters called theCode and theKind. A native driver is reentrant to the extent that during any call from the driver to IOCommandIsComplete the driver may be reentered with another request. A native device driver does not have any sort of header. It will however, export a data symbol called TheDriverDescription. A driver uses this data structure to give header like information to the Device Manager. The Device Manager 90 uses the information in TheDriverDescription to set the dCtlFlags field in the driver's DCE. A native device driver cannot make use of the dCtlEMask and dCtlMenu fields of its driver control block. Native drivers 80 are not used for creating desk accessories.

Detailed Description Text (29):
A device driver presents the operating system 30 with a self-describing data structure called a driver description ("DriverDescription") 80a. As shown below with respect to a particular embodiment, the driver description ("DriverDescription") 80a is used by matching mechanism of the DLL 45 of the present invention to (1) match devices to drivers; (2) identify devices by category of functionality; (3) provide driver name information; (4) provide driver version information; (5) provide driver initialization information; and (6) allow replacement of currently installed driver. By providing a common structure to describe drivers 80, the operating system 30 is able to regularize the mechanisms used to locate, load, initialize, and replace them. The operation system 30 treats any code fragment that exports a DriverDescription structure as a driver within the present invention. The structure DriverDescription 80a is used to match drivers with devices, set up and maintain a driver's runtime environment and declare a driver's supported services. An exemplary structure is shown below wherein the "driver name" 80c (FIG. 5) is located within the driverType information:

Detailed Description Text (60):
When the Device Manager 95 receives a KillIO request, it removes the specified parameter block from the driver I/O queue. If the driver responds to any requests asynchronously, the part of the driver that completes asynchronous requests (for example, an interrupt handler) might expect the parameter block for the pending request to be at the head of the queue. The Device Manager 95 notifies the driver of KillIO requests so it can take the appropriate actions to stop work on any pending requests. The driver must return control to the Device Manager 95 by calling IOCommandIsComplete.

Detailed Description Text (62):
Under certain conditions, it may be desirable to replace an installed driver 80. For example, a display card may use a temporary driver during system startup and then replace it with a better version from disk once the file system is running and initialized. Replacing an installed driver is a two-step process. First, the driver to be replaced is requested to give up control of the device. Second, the new driver is installed and directed to take over management of the device. In one embodiment, two native driver commands are reserved for these tasks. The kSupersededCommand selector tells the outgoing driver to begin the replacement process. The command contents are the same as with kFinalizeCommand. The outgoing driver should take the following actions: (1) If it is a concurrent driver, it should wait for current I/O actions to finish. (2) Place the device in a "quiet" state. The definition of this state is device-specific, but it can involve such tasks as disabling device interrupts. (3) Remove any installed interrupt handlers. (4) Store the driver and the device state in the Name Registry as one or more properties attached to the device entry. (5) Return noErr to indicate that the

driver is ready to be replaced. (6) The kReplaceCommand selector tells the incoming driver to begin assume control of the device. The command contents are the same as a kInitializeCommand.

Detailed Description Text (65):
In one embodiment, families can be stored in several locations within the computer system 120. A family can be located in RAM 102 or in a file system (e.g. on a disk drive 104). In addition to matching a device to appropriate drivers, the present invention automatically matches a device of the device tree 10 with a corresponding family. A family consists of code which, when executed, provides a universal interface for software, such as an application program, to interface with the selected device driver of that family. Families include storage devices, sounds, display devices, etc.

Detailed Description Text (71):
A family presents the operating system 30 with a self-describing data structure called a family description ("FamilyDescription") 90a. As shown below with respect to a particular embodiment, the family description ("familyDescription") 90a is used by matching mechanism of the DLL 45 of the present invention to match devices to families. By providing a common structure to describe families 80, the operating system 30 is able to regularize the mechanisms used to locate, load and initialize them. The operation system 30 treats any code fragment that exports a FamilyDescription structure as a family within the present invention. The structure family description 90a is used to match families with devices, set up and maintain a family's runtime environment and declare a family's supported services. An exemplary structure is shown below wherein the family name, or category name 90e (FIG. 5b) is located within the familyType information.

Detailed Description Text (75):
FIG. 7 illustrates functions of the DLL 45 of the present invention. At logic block 200, the present invention performs driver matching and family loading which includes determination of a particular driver and family for a particular device of the device tree 10 database. In order to perform this, logic block 200 interfaces with the name registry (also called device tree 10 database) and also with RAM unit 102 and ROM unit 103. Drivers and families may be located in these memory units. The file storage 104 (where drivers may be located in the device driver and families folder) is also coupled to communicate with the logic block 200. The CFM 40 is also coupled with block 200. The DLL 45 performs driver matching and family loading in block 200 as well as device installation 210 once an appropriate driver is selected for a device. At block 220, information retrieval is performed which allows a user to retrieve particular information about a recognized driver. Also, the DLL 45 of the present invention performs driver removal at block 240. These functions will be discussed in further detail to follow.

Detailed Description Text (76):
FIG. 8 illustrates a flow diagram of processing performed by the DLL 45 of the present invention for automatically matching device drivers 80 to a particular identified or selected device of the devices reported in the device tree 10 database using candidate lists and sorting. This process is repeated for each device of the device tree 10 starting from the top and continuing downward through the tree 10. Processing 200 is invoked upon a request to locate a driver and associated family for a given or "selected" device. Processing logic 200 starts at block 305 wherein a candidate list is constructed and is associated with the selected device. This list contains a grouping of those drivers having a driver name that matches with the given device's device name or compatible device name. These drivers represent a set of drivers that perhaps will properly configure the given device. FIG. 9a illustrates the steps performed by logic block 305 in more detail.

Detailed Description Text (77):

With reference to FIG. 8, after a candidate list is constructed, the present invention flows to logic block 310 wherein the candidate list is sorted by priority ranking as to which members of the candidate list are more than likely to match with the selected device and those drivers that are less likely to match with the selected device. The steps performed by logic block 310 are further described with reference to FIG. 10. With reference to FIG. 8, after a candidate list is sorted by priority ranking, the present invention flows to logic block 600 wherein DLL determines the list of families required and installs them. Exemplary steps performed by logic block is shown in FIG. 9b. With reference to FIG. 8, at block 315 the present invention determines family requested to the DLL 45 to actually install a driver to the selected device. If installation request is made, the DLL installs the requested drives in block 210.

Detailed Description Text (78):
At block 330, the present invention instructs the operating system 30 to scan the devices of the computer system 120 to determine if all of the devices that the selected device needs to operate (the "parent devices") are present within the computer system 120. If the parent devices are present, then the selected driver can be installed. Since devices of the device tree 10 database are processed through the DLL 45 of the present invention from top to bottom, the parent devices for a particular selected device should be operational (e.g., processed through blocks 200 and 210) before the selected device is processed by the present invention. If the required parent devices are not operational yet or not present, then block 210 is avoided. If the required resources are available, then at block 210 the present invention performs an installation wherein the determined driver is installed with respect to the selected device and the device becomes active. It is appreciated that the processing of logic blocks 200 and 210 are typically performed at initialization for each device of the device tree 10 database, starting with the top node and working down the device tree 10 so that parent devices are configured first before their child devices are configured.

Detailed Description Text (79):
With reference to FIG. 9a, a flow diagram is illustrated describing the logic steps of the logic 305 of the present invention DLL 45 in constructing a candidate list of drivers for the selected device. Logic 305 starts at block 410 wherein pertinent information regarding the device nodes of the device tree 10 database are accessed for the particular device. If the particular device has an associated driver within its node of the device tree 10 (e.g., a "default driver"), processing continues because this default driver can become replaced by an updated driver depending on the priority of the drivers in the candidate list built for the selected device. At block 410, the present invention obtains the following properties: (1) the device name 50; and (2) the compatible names 60a of the selected device (see FIG. 4) located within the compatible property 60. After the information of logic block 410 is accessed, the present invention at logic block 420 then access the available drivers recognized in the system to construct a first set of drivers. These drivers may reside in the device tree 10 database, in the ROM 103, in RAM 102 and in the extensions folder (e.g., device driver folder) of the disk drive 104. At block 430, the present invention selects a first driver for comparison. This driver is the "given driver." At block 430, the candidate list for the selected device is then cleared so the new list can be created.

Detailed Description Text (80):
At logic block 440, the present invention examines the DriverDescription 80a information for this given driver to determine the driver name 80c (FIG. 5) of the given driver which is stored in the DriverDescription, in one embodiment, as the "nameinfostr field" of the deviceType structure for the given driver. Importantly, at logic block 440, the present invention performs a comparison between: (1) the device name 50 of the selected device and the driver name 80c of the given driver; and also between (2) each compatible name 60a of the selected device against the driver name 80c of the given driver. If there is a match between (1) or (2) above,

then the match characteristics are recorded (e.g., was it match by (1) or by (2)) and at logic block 450, the given driver is added to the candidate list for the selected device if a match in 440 happened. It is appreciated the candidate list can be stored in RAM 102. The processing of the present invention then flows to logic block 460. If there was not a match at block 440, then the present invention flows to block 460 without adding the given driver to the candidate list.

Detailed Description Text (81):
At block 460, the present invention determines if the given driver is the last driver of the drivers discovered in block 420 (e.g., those drivers recognized by the computer system 120). If so, then the process 305 exits from block 460. If not, then logic block 460 is exited and processing flows to block 470 wherein the present invention selects the next driver of the set of drivers discovered in block 420. This next driver is then the "given driver" and processing returns to block 440 to determine if this next given driver should be added to the selected device's candidate list. As with any driver, the next driver selected at block 470 can reside in RAM 102, in an extension ROM 103 within the device tree 10, or within a file on the disk 104. At the completion of process 305, an unsorted candidate list is then constructed for the selected driver and is stored in RAM 102.

Detailed Description Text (82):
With reference to FIG. 10, the logic steps of logic block 310 are illustrated. Block 310 performs the candidate list sorting for a particular candidate list associated with the selected driver. At block 460, the candidate list for the selected device is accessed in RAM 102. This is a candidate list generated by logic 305 and is particular to the selected driver. At block 470, the present invention sorts the candidate list and places those drivers at the top or head of the candidate list that have a driver name 80c that matched with the device name 50 of the selected device. At the same time or sequentially, block 480 resolves any prioritization ties by using version information stored in the driver description between drivers of the candidate list. Those drivers with a more appropriate version (e.g., highest version or version closest to the selected driver) are placed higher in the candidate list priority. Block 470 then places in lower priority those drivers having a driver name 80c that matched with the selected driver's compatible names 60a. Again, version information (in logic block 480) with respect to these drivers is used to perform prioritization the same as discussed above. At the completion of block 480, the candidate list for the selected device is sorted by priority of most likely for validation (e.g., most likely to be compatible with the selected device).

Detailed Description Text (83):
With reference to FIG. 9b, the logic steps of logic block 600 of the present invention are illustrated. Block 600 performs a procedure to find suitable families for the matching drivers and installing them. Once the families are installed, the installation of drivers in the sorted candidate list is attempted. At logic step 610, the present invention accesses the sorted candidate list for a particular device. At block 620, the present invention accesses or "gets" the first driver of this candidate list. At block 630, the present invention reads the category information from the driver descriptor 80a. At block 640, the present invention checks if the category is already required by the drivers processed previously. If the test return "yes", the present invention flows to block 660 otherwise it goes to block 650. At block 650, the present invention adds the category information to the list of categories required in flows to block 660. At block 660, the present invention determines if the selected driver was the last driver of a selected candidate list of the selected device. If not, the processing flows to block 670 wherein the present invention selects the next driver in sequential order from the sorted candidate list of the selected device. Processing then flows back to block 630.

Detailed Description Text (84):

Preferably, when all the drivers in the selected list are processed, processing flows to block 680 wherein the present invention finds a matching family either in RAM unit or 102 or the file storage 104. During the matching in block 640, the DLL compares the category information read in block 650 to the category name of the family 90a. If more than one family for the same category is present, DLL selects the family with the higher version number in 90a (FIG. 5b). At block 690, the present invention loads and installs the families found in block 680. During the installation phase, families attempt to install the best driver candidate for the particular device. The steps performed to select the best driver are further defined in FIG. 9c.

Detailed Description Text (85):
With reference to FIG. 9c, the logic steps of logic block 690 of the present invention are illustrated. Block 690 performs a procedure of attempted installation of the drivers of the candidate list for the particular device (e.g., a trial and error approach based on the prior information compiled by the present invention). At logic step 510, the present invention accesses the sorted candidate list for the particular device. At block 520, the present invention accesses or "gets" the first driver of this candidate list. At block 530, the present invention attempts to install this selected driver with the selected device to validate the match. The selected driver validates the match by probing the device and performing some diagnostic operations between the selected driver and the device. Any number of different diagnostic operations can be used within the scope of the present invention for this step. Assuming the selected driver is appropriate, at logic block 540, the selected driver confirms the validation by returning a "no error" status flag to the DLL 45. If an error status was returned, or the "no error" status fails to return, then processing flows to logic block 570 wherein the present invention determines if the selected driver was the last driver of the selected candidate list for the selected device. If not, processing flows to block 560 wherein the present invention selects the next driver in sequential (e.g., priority) order from the sorted candidate list of the selected device. Processing then flows back to block 530 to determine if the next driver will validate the match with the selected device.

Detailed Description Text (86):
At block 570, if the selected driver happens to be the last driver of the sorted candidate list, then at block 580, the present invention returns an error code indicating that no compatible driver could be found for the selected device. Returning to block 540, the present invention flows to logic 550 if the selected driver did indeed match with the selected device. At block 550, the selected driver's category information is then compared against the category information of the selected device which is stored in the device tree 10 as property information. If no match is performed between the categories, then the match is not validated, so processing returns to logic block 570. If the categories match, then at block 555, the selected driver is said to be determined and a proper validation of the match occurs between it and the selected device.. This information is then returned from block 320.

Detailed Description Text (87):
The operating system 30 can utilize the above logic of the present invention at various times, but in one embodiment it is used during the boot phase and at any time a new device is added to the device tree 10 (which can also add new drivers to the available list of drivers used by the present invention). It is appreciated that as the system boots and new devices are configured and "wake up" and are added to the device tree 10 (by IEEE P.1275), it is possible for a device to be assigned a driver via the present invention and then re-assigned a newer or more appropriate driver later as they become available during the boot phase. In other words, drivers located on the hard disk are not available until the hard disk itself becomes configured as a device. In such case, the scope of drivers available at block 420 (FIG. 9a) of the present invention is dynamic during the boot phase and

will increase as soon the as hard drive is properly configured. In this example, a particular device can be initially configured with a driver from ROM and subsequently can be reconfigured with a more appropriate driver from the driver folder of the hard drive because the new driver will become higher in the candidate list (over the old driver) upon subsequent processing of the particular device by the present invention. Therefore, the candidate lists for a particular device are dynamic in that they will grow depending on the set of drivers that are available within the computer system 120 and recognized by the present invention.

Detailed Description Text (121):
Given a pointer to a CFM file system specification, GetDriverDiskFragment uses the CFM search path to find and load a driver code fragment. It returns the loaded driver's CFM connectionID value in fragmentConnID, a pointer to its DoDriverIO entry point in fragmentMain, and a pointer to its Driver Description in theDriverDesc.

Detailed Description Text (129):
Given a pointer to the RegEntrvID value of a device, FindDriversForDevice finds the most suitable driver for that device. If the driver is located in a file, it returns a pointer to the driver's CFM file system specification in fragmentSpec and a pointer to its Driver Description in fileDriverDesc. If the driver is a fragment located in memory, FindDriversForDevice returns a pointer to its address in memAddr, its length in length, its name in fragName, and a pointer to its Driver Description in memDriverDesc.FindDriversForDevice initializes all outputs to nil before searching for drivers.

Detailed Description Text (169):
Applications wishing to remove an installed driver can use RemoveDriver, see block 240 of FIG. 7.

Detailed Description Text (176):
GetDriverInformation returns a number of pieces of information about an installed driver, see block 220 of FIG. 7.

Detailed Description Text (196):
DeviceProbe is used to determine if a hardware device is present at the indicated address. This process can operate during block 530 of FIG. 11.

Detailed Description Text (201):
In one embodiment, the second phase of startup comes after the file system is available. In this second phase the device driver folder (e.g., extensions folder) is scanned for Family Experts, which are run as they are located. Their job is to locate and initialize all devices of their particular service category in the Name Registry 10. The Family experts are initialized and run before their service category devices are initialized because the Family expert extends the system facilities to provide services to their service category devices. For example, the DisplayManager extends the system to provide VBL capabilities to `disp`service category drivers. In the past, VBL services have been provided by the Slot Manager; but with native drivers, family-specific services such as VBL services move from being a part of bus software to being a part of family software.

Detailed Description Text (204):
If there is a driver in ROM 103 for a device, the driver, AAPL, MacOS, PowerPC property is available in the Name Registry 10 whenever a client request is made to use that device. However, after the operating system is running and the file system is available, the ROM driver may not be the driver of choice. In this case, the ROM-based driver is replaced with a newer version of the driver on disk. In the system startup sequence, as soon as the file system 104 is available the operating system 30 searches the device driver folder (e.g., Extensions folder) and matches drivers in that folder with device nodes in the Name Registry. The driverInfoStr

and version fields of the DriverType field of the two DriverDescriptors 80a are compared, and the newer version of the driver is installed in the tree. When the driver is updated, the DriverDescriptor property and all other properties associated with the node whose names begin with Driver are updated in the Name Registry 10.

Detailed Description Paragraph Table (5):
______________________________ struct DriverServiceInfo { OSType serviceCategory; OST,vpe serviceType; NumVersion serviceVersion; }; typedef struct DriverServiceInfo DriverServiceInfo; typedef struct DriverServiceInfo *DriverServiceInfoPtr; enum { /*used in serviceCategory*/ kServiceCategoryDisplay = 'disp', /*display*/ kServiceCategoryopentransport = 'otan', /*open transport*/ kServiceCategoryblockstorage = 'blok', /*block storage*/ kServiceCategorySCSISim = 'scsi', /*SCSI SIM*/ kServiceCategoryndrvdriver = 'ndrv' /*generic*/ }; Field descriptions: serviceCategory Specifies driver support services for given device family. Some examples of device families are: Name Supports services defined for: 'blok' block drivers family 'disp' video display family 'ndrv' gneric native driver devices 'otan' Open Transport 'scsi' SCSI interface module serviceType Subcategory (meaningful only in a given service category). serviceVersion Version resource ('vers') used to specify the version of a set of services. It lets interfaces be modified over time. ______________________________

Detailed Description Paragraph Table (6):
______________________________ OSErr DoDriverIO (AddressSpaceID) spaceID IOCommandID ID, IOCommandContents contents, IOCommandCode code, IOCommandKind kind); typedef KernelID AddressSpaceID; spaceID The address space containing the buffer to be pre- pared. Mac OS 7.5 provides only one address space, so this field must be specified as kCurrentAddressSpaceID. ID CommandID contents An IOCommandContents I/O parameter block. Use the InitializationInfo union member when calling to initialize the driver, FinalizationInfo when removing the driver, DriverReplaceInfo when replacing, DriverSupersededInfo when superseding, and ParmBlkPtr for all other I/O actions. code Selector used to determine I/O actions. kind Options used to determine how I/O actions are per- formed. The bits in this field have these meanings: Bit Meaning 0 synchronous I/O 1 asynchronous I/O 2 immediate I/O ______________________________

Detailed Description Paragraph Table (14):
______________________________ OSErr DoKillIOCommand (ParmBlkPtr thePb) { /* Check internal queue for request to be killed; if found, remove from queue and free request */ return noErr; } /* Else, if no request located */ return abortErr; thePb Pointer to a Device Manager parameter block

____

Detailed Description Paragraph Table (18):
______________________________ OSErr GetDriverDiskFragment (FSSpecPtr theFragmentSpec, ConnectionID *fragmentConnID, NuDriverEntryPointPtr *fragmentMain, DriverDescriptionPtr theDriverDesc); fragmentSpec pointer to a file system specification fragmentConnID resultingCFM connectionID fragmentMain resulting pointer to DoDriverIO driverDesc resulting pointer to DriverDescription

____

Detailed Description Paragraph Table (25):
______________________________ OSErr FindDriversForDevice (RegEntryIDPtr device, FSSpec *fragmentSpec, DriverDescription *fileDriverDesc, Ptr *memAddr, long *length, StringPtr fragName, DriverDescription *memDriverDesc); device device ID fragmentSpec pointer to a file system specification fileDriverDesc pointer to the Driver Description of driver in a file memAddr pointer to driver address length length of driver code fragName name of driver fragment memDriverDesc pointer to the Driver Description of driver in ______________________________ memory

Detailed Description Paragraph Table (39):
____________________________________ OSErr InstallDriverFromFile (FSSpecPtr fragmentSpec, RegEntryIDPtr device, UnitNumber beginningUnit, UnitNumber endingUnit, refNum *refNum); fragmentSpec pointer to a file system specification device pointer to Name Registry Specification beginningUnit low unit number in Unit Table Range endingUnit high unit number in Unit Table Range refNum resulting Unit Table refNum ____________________________________

L27: Entry 4 of 5 File: USPT Jul 11, 2000

DOCUMENT-IDENTIFIER: US 6088706 A
TITLE: System and method for managing replicated data by merging the retrieved records to generate a sequence of modifications

Detailed Description Text (4):
Ideally, mobile communication would be handled on a layer low enough to allow a large number of applications to benefit and high enough to give insight into the type of data being transmitted to allow the use of specialized compression and reduction methods. According to the preferred embodiments of the present invention, mobile communication is handled at the file-system level, since most application programs use files for data input and output, so a broad spectrum of applications can benefit. Files also comprise data-units rather than pieces of data, and thus, the file-type can often be inferred. Furthermore, it has been realized that continuous connection is not required during normal operation, since all applications can operate on data available locally, thus allowing reasonable support for interactive applications; connections are only required occasionally to re-synchronize the files.

Detailed Description Text (5):
In the area of wired (`terrestrial`) communications networks, as opposed to mobile communications networks, some distributed filing-systems have been developed. Network file systems such as Sun Microsystems Inc's Network File System (NFS), or the Andrew File System (AFS) from Transarc Corp, are used on wired networks to gain access to data files held at remote nodes. Thus, local applications can work on data held at remote sites and files can be shared between many users. Effectively, the use of a network is entirely hidden from the user, who sees all files as local. In the simplest form, these systems operate by redirecting read/write operations across the network using some sort of remote procedure call facility. Caching is often used to reduce traffic load on the network.

Detailed Description Text (6):
When simultaneous editing is allowed, conventional file systems make use of various locking methods to ensure data consistency. Voting has been proposed instead of locking, but write-access is still restricted to one site which has to obtain write-permission for sufficiently many copies prior to accessing a file. Many systems use tokens to coordinate access to replicated files, but the passing of tokens again requires communication links to be operational between the sites. In the article "Consistency and recovery control for replicated files", Proceedings of the 10th ACM Symposium on Operating Systems Principles, December 1985, Davcev and Burkhard have proposed a system which allows write-access when the network is partly disconnected, but only within the so-called `majority-partition`. In the article "An overview of reliability mechanisms for a distributed data base system", Proceedings of the spring COMPCON, February 1978, Hammer and Shipman have proposed a technique which does not require locks for write-operations and therefore allows files to diverge slightly but relies on the communication links to resolve the resulting inconsistencies within tight time constraints.

Detailed Description Text (7):
Lotus Notes allows multiple read/write replicas of its special database. Replicas are periodically reconciled, usually no more than once or twice a day. Detection of

a conflict between replicas causes the creation of separate versions with no attempt to automatically resolve the conflicts, this resulting in significant manual burden whenever a conflict occurs. A different approach is discussed in European Patent Application EP-A-0,684,558, which describes a replication system in which a plurality of servers maintain updatable replicas of a file. An update propagation protocol is used, which is described as "aggressive" in that it causes a replica update as soon as possible after a failure leading to inconsistent data has been identified. In effect, the servers coordinate amongst themselves to detect replica inconsistencies and initiate an update protocol to detect stale or conflicting replicas without waiting for a client request for data. Although some conflicts are resolved automatically, manual intervention may be required to repair conflicting

Detailed Description Text (10):
The above file system level techniques are not suitable for a mobile environment due to their reliance on fast, continuously available communication links and/or their liberal use of locking methods which seriously hampers prolonged periods of disconnected operation.

Detailed Description Text (11):
FIG. 2 illustrates the set-up of the data management system of the preferred embodiment, which will be referred to hereafter as the `Mobile Application Framework` 200. In the FIG. 2 example, both users 210, 220 operate locally with local copies of the shared file (represented by the disk-symbol 205), while the framework 200 underneath endeavours to keep both copies synchronised. It is important to note that with this framework it is no longer the applications that communicate or initiate transmissions, but the underlying framework.

Detailed Description Text (12):
Some file systems, notably `CODA` (see the article "Disconnected Operation in the Coda File System", ACM Transactions on Computer Systems, 10(1), February 1992, by J Kistler and M Satyanarayanan), are now being extended to allow disconnected operation for periods of networks being down and thus make a step in this direction. Additionally, the article "Combining Location and Data Management in an Environment for Total Mobility" by Monica Wachowicz and Stefan Hild, Proceedings of the International Workshop on Information Visualization and Mobile Computing, Rostock, Germany, February 1996, describes a `total mobility` architecture, in which a user no longer carries his portable computer with him/her but instead will register with a rented computer at his/her destination. As part of this architecture, a disconnected operation is contemplated, in which data is manipulated in disconnected mode by applications running on the mobile host. Changes to the data file are stored to facilitate later reconciliation with other copies of that file. A first outline of the data management subsystem forming part of the total mobility architecture was presented by Stefan Hild in the position statement "Disconnected Operation for Wireless Nodes", published in the Proceedings of the ECOOP '95 Workshop on Mobility and Replication, European Conference on Object Oriented Programming, August 1995. This paper briefly describes the general concept of disconnected file access and reconciliation within a mobile environment.

Detailed Description Text (13):
The `Mobile Application Framework` takes the view that disconnected operation (i.e. no connection is established with the stationary host) is the norm and that connected periods are the exceptions, rather than vice versa. Consequently, the `Mobile Application Framework` of the preferred embodiment differs in many ways from conventional network file system and those allowing disconnected operation, as will become more apparent from the more detailed description which follows.

Detailed Description Text (14):
In preferred embodiments, the `Framework` is used as a tool for sharing a small

number of files of primary importance between a small number of users. Setting up such a `work-group` is a simple but conscious process. By taking this careful approach to the concept of `sharing`, the `Framework` can afford to take a much loser stance on file consistency. Hence, the `Framework` is neither a replacement nor an extension to conventional network file systems but facilitates the management of replicated files. Some of its main features will now be discussed in more detail.

Detailed Description Text (19):
FIG. 4 illustrates the internal components of the Framework and also gives some justification for its name. Rather than providing a platform for a mobile application to run on, it also consists of components which run on the same level as the application (especially the logger 410) and thus provides a `framework` for the application to run in. In detail, the different components are the logger 410, which logs all modifications executed on the local copy of the shared data file, the file being represented by the disk 420 in the figure. The logger 410 has available a number of models 430 which are formal descriptions of contents-type and possible edit-operations for various file-types to assist the detection of modifications. Defaults are available which have been designed to work with any file-type. These models will be discussed in more detail later.

Detailed Description Text (31):
Firstly, it is necessary to assign a priority to each modification so that, given two conflicting modifications, one will overwrite the other. This priority assignment is typically based on the location of the file copy; some file-systems which allow disconnected operation define a `master-copy` which, in the case of any conflicts, has priority over all other copies. This approach is not taken in preferred embodiments since it is believed that users are likely to change between machines in which case it would not be sensible to bind the priority of a modification to the location at which it was executed. Instead, the preferred embodiment assigns priority-levels based on the identity of the user who modifies the file. The time-stamps recorded with the modifications are also used to assign priorities. The view that early modifications over-rule later ones is preferably adopted, since the opposite policy would lead to the counter-intuitive situation that modifications have a greater chance of committing successfully the later they are executed.

Detailed Description Text (46):
Current network costs and availability. Conditions to trigger reconciliations may be relaxed during off-peak hours while network connections are available at a cheaper rate or while cheaper networks are available. For example, it is conceivable that the node may actually be connected to a free wired network connection for certain periods of time so reconciliations can be executed permanently. Then, the Framework effectively operates like a conventional `Network File System`. Changes are propagated immediately to all copies and conflicts are consequently unlikely. In the other extreme, reconciliation steps may be delayed if mobile data channels are detected to be of poor quality, resulting in high error rates, many retransmissions and consequently high transmission costs.

Detailed Description Text (49):
The above description describes the Mobile Application Framework of the preferred embodiment, this being a generic system which allows standard applications to make use of mobile data links in a budgetable and controllable manner. This is achieved by adding support for disconnected operation on file system level. The problem of diverging files due to unrestricted access during long period of disconnected operation is solved by using loggers which create detailed modification histories for each replicated file, allowing fully automatic conflict resolution. Although a generic algorithm can never be expected to produce optimal results in all imaginable cases, the resulting files could always be worked with and cleaned up manually if necessary.

Detailed Description Text (51):
The positioning of the Framework at the file-system level allows the communication sub-system to make use of type information which can be inferred easily by applying specialised compression and reduction methods to files prior to transmission.

Other Reference Publication (2):
Kumar, "Coping with Conflicts in an optimistically Replicated File System", IEEE, pp. 60-64, Mar. 1990.

Other Reference Publication (8):
"Using Briefcase to Keep Documents Up-To-Date", "To Keep Files Synchronized by Using a Floppy Disk", "To Synchronize Files on Connected Computers", from Help files for Microsoft Briefcase in Microsoft Windows 95.

Other Reference Publication (11):
"Disconnected Operation in the Coda File System", James Kistler and M. Satyanarayanan, ACM Transactions on Computer Systems, vol. 10, No. 1, Feb. 1992, pp. 3-25.

Previous Doc Next Doc Go to Doc#